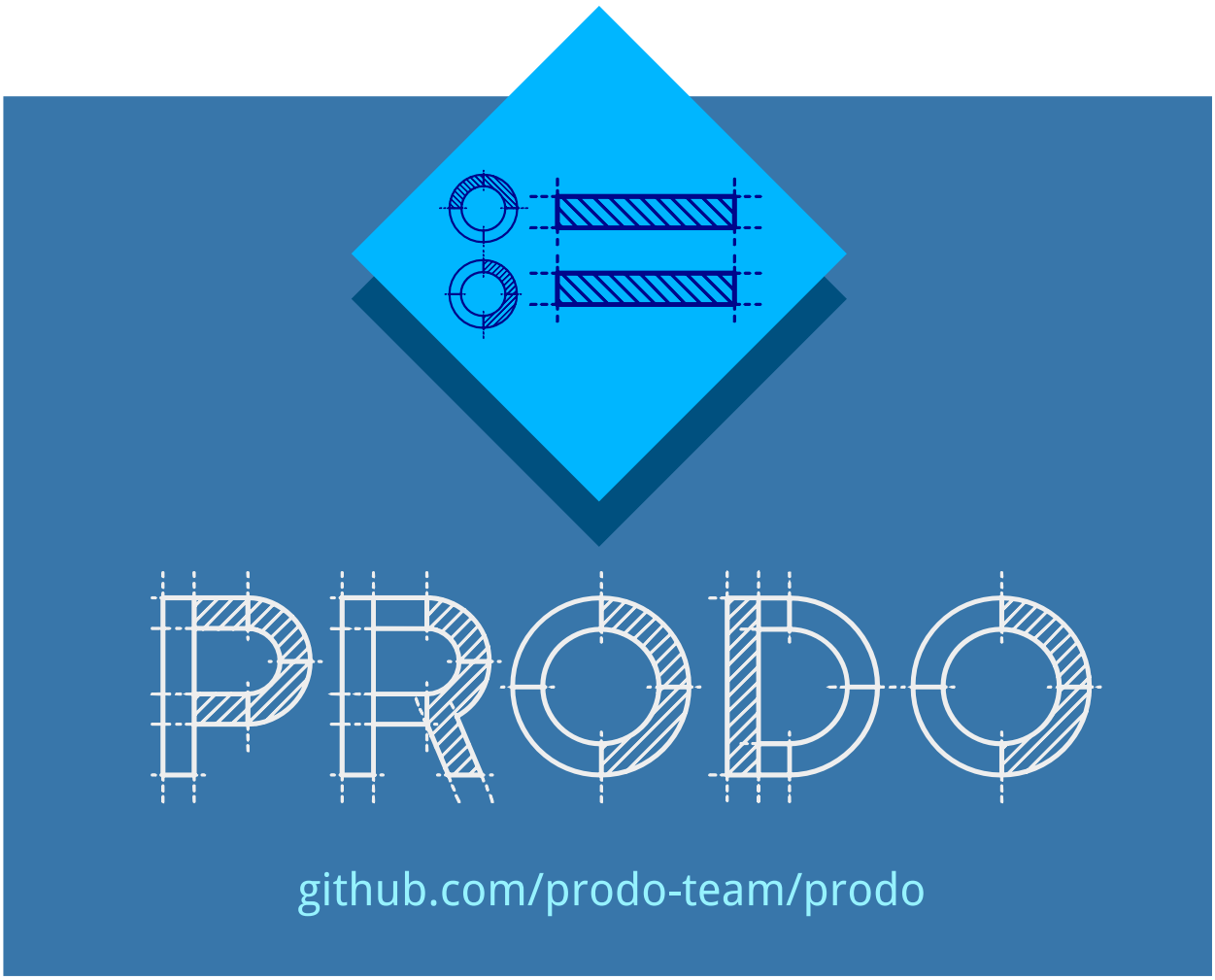

THE PRODO MANUAL

Introduction

This manual is for beginners, for users who are new to Prolog. It assumes that you have a basic knowledge of programming. And if you know Python, Ruby, C, or Javascript, that would be great! Those languages and their close relatives serve as the overall inspiration for *Prodo*: a simple, strongly typed imperative programming language.

Using the Interpreter

If you're reading this, you've probably already downloaded the necessary files for running *Prodo* code. But if you haven't, you can follow these steps to set up:

1) Get the dependencies
    - **Python.** The *Prodo* parser you'll be using is written in Python. You must download and install Python in order to use the *Prodo* parser. If you don't have it already, go to http://python.org/ to get it now.
    - **YAPPS.** You will also need the YAPPS Python module. After you've installed Python and added it to your PATH, enter this into your command line to install YAPPS and its dependencies:

```
pip install 'git+https://github.com/mk-fg/yapps.git#egg=yapps'
```

      For more information about YAPPS, visit its GitHub page: https://github.com/smurfix/yapps

2) Get Prodo. Once all the dependencies are installed, you will need the *Prodo* parser. (If you already have the files, skip this step) This is a collection of files that scan, parse, and run the Prodo code that you have written. It also includes the *Prodo Python Library*, which is a bunch of built-in utilities and functions that come with *Prodo*.
    - **Download Prodo.** Get the latest release of Prodo, which are simply a collection of grammar, python, and Windows batch files. You can download them here: https://github.com/prodo-team/prodo/releases/
      After downloading the code, put them in any folder you like. You may also want to add that folder into your system's PATH variable for easier access.
3) Write Your Code. Assuming you already know the Prodo language, you can start cranking out code on any text editor of your choosing. Save your files with the *.prodo extension.
4) Run the Parser.
    a. **For Windows Users:**
        i. Open a command prompt window.
        ii. Navigate to the directory where the Prodo parser files are located. (If you included the directory in your PATH variable, you don't need to do this).
        iii. Run the Prodo batch file by entering this in your command line:

```
prodo mycode.prodo
```

where your code's filename is "mycode.prodo". It can be an absolute or relative filename, depending on where your file is located.

b. **For Other Systems (Linux, OS X):**
   i. Open a terminal window.
   ii. Navigate to the directory where the Prodo parser files are located. (If you included directory in your PATH variable, you don't need to do this).
   iii. Execute the Prodo bash shell script by entering this in your command line:

```
./prodo.sh mycode.prodo
```

where your code's filename is "mycode.prodo". It can be the relative or absolute path, depending on your configuration.

Debugging

If your code contains errors, the parser will return one of the following types of error messages.

1) Syntactic Errors – your code's syntax does not follow the Prodo specifications. The line number and the missing tokens will be pointed out to you.
2) Name Errors – you tried to use an undeclared variable or undefined subprogram. This is also the error triggered when you try to pass an incorrect number of arguments to a subprogram.
3) Structure Member Not Found – you tried to access a non-existent member in a structure.
4) Index Errors – you tried to use an illegal index to access an array's element. Array indices must evaluate to integers.
5) Illegal Value – you tried to use an unusable value or convert an inconvertible value.
6) EOF Errors – the parser was looking for more input but couldn't find it anywhere. This can happen if a read( ) instruction receives absolutely no input.
7) Assignment Errors – you tried to assign a value of a different type to a variable without an explicit cast. The non-matching types are displayed for you.
8) Argument Errors – you passed arguments with incorrect types to a subprogram. The non-matching types are displayed for you.
9) Conclusion Errors – a subprogram is trying to return a value with a type that doesn't match its declared return type.

10) Logical Operator Error – you tried to use the ‘and’, ‘or’, or ‘xor’ operator(s) on non-Boolean expressions.
11) Type Errors – you tried to use a value with an invalid type in an assignment or operation. This may occur if you try to add strings to integers without a cast, or try to use a string as a loop iterator.
12) For Loop Increment Error – you tried to use zero as a for loop increment, i.e. argument to the “by” clause in a for-loop. A valid increment must be non-zero.
13) File Read Errors – there was an error reading from a file. You might encounter this when using the f_read subprogram.
14) File Write Errors – there was an error when writing into a file. You might encounter this when using the f_write subprogram.

Note that *Prodo* is interpreted, so the errors may not be raised unless that part of the code is run. Syntactic Errors are always detected, however.

Prodo Cheat Sheet

Here’s a quick summary of how to do things in *Prodo*. For more in-depth information, please look at the Prodo Language Functional Specifications.

Syntax	Action/Meaning	Remarks
`type variable := value`	Declare a variable with a specified type and initialize to value	A variable declaration is the **only** time when the value will be automatically cast (because the type is already explicitly specified)
`type a, b, c := value`	Declare 2 or more variables with a specified type and initialize to a common value	The newly initialized values will have the same type and will be initialized to the same value
`a := b`	Assign b to a	b and a must be of the same type
`{1,3.14,”a”}`	An array literal with elements 1 (int), 3.14 (real) and “a” (str)	An array in *Prodo* is a heterogeneous list
`my_array[i]`	Access element of array at index i	Index must be within range of indices of that

		array. Negative indices loop around from the end of the array.

`structure my_struct`
`  int x := 8`
`  real y := 5.2`
`  str z := "cheese"`
`  …`
`end`
	Declare a structure *my_struct* with members x, y, z,…	Members can only be variables, and must be initialized.

`my_struct.x`	Access a member *x* of a structure *my_struct*	Structures cannot be nested
`a + b * c / d % e`	Do basic math	

`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
	Operate on *a* and *b* and assign the result to *a*	a and b must have the same type.

`a++` `a--`	Increment/Decrement a by 1 or 1.0	a must be an int or real variable. These **must appear on their own lines**
`-a`	Negate the value of a	a must be an int or real value
`{1,2,3} + {"apple", "orange"}`	Concatenate arrays	Results to {1,2,3,"apple", "orange"}
`"Hello " + "World"`	Concatenate strings	Results to "Hello World"
`"etc" * n`	Repeat a string *n* times	n must be a positive integer. If it is zero or negative, the expression evaluates to an empty string
`type(var)`	Cast *var* into a *type* value	**Prodo does not allow coercion** except when declaring variables
`[a + b]*c`	Group an arithmetic expression	The most deeply nested groupings will have the highest priority
`for | int k := 1 to 10 |`	Loop inside the open interval [1, 10) with an increment of 1	Only int and real can be used as a loop counter (to prevent floating point accuracy errors) The loop counter can be declared outside.

`for | int k := 1 to 10 by 2 |`	Loop inside the open interval [1, 10) with an increment	The increment can also be a negative, but cannot be zero.

`while | Boolean |`
`  statements`
`end`
	While loop (logic-controlled iteration with pre-test); loop while the Boolean is true	

`loop`
`  statements`
`end | Boolean |`
	Logic-controlled iteration with post-test; Loop once, then check the Boolean for succeeding loops	Guaranteed to run at least once

`if | Boolean |`
`  statements`
`end`
	Single selection statement	Only the if clause is required. It must come before all other clauses

`if | Boolean |`
`  statements`
`else`
`  statements`
`end`
	Two-way selection statement	The else clause has no Boolean. It must be the last statement.

`if | Boolean |`
`  statements`
`elseif | Boolean |`
`  statements`
`elseif | Boolean |`
`  statements`
`end`
	Multiple-way selection statement	Zero or more elseif clauses can be added. They must precede the else clause. If an earlier if/elseif clause has been executed, the remaining ones are ignored

`a == b, a != b, a < b` `a > b, a <= b, a >= b`	Relational expressions	

`a == b and b != c`
`a == b or b != c`
`a == b xor b != c`
`not (a == b)`
	Logical AND
	Logical (inclusive) OR
	Logical (exclusive) OR
	Logical NOT
		Boolean expressions **cannot be grouped** to simplify the syntax. Since Boolean operators have no real order of precedence, they are evaluated from left to right, in the order that they appear

`~ Comment`	A comment	Everything after the ~ until after a new line is ignored.

`fcn type name (params)`
`  statements`
`end`
	Define a function with return type ‘type’, name ‘name’ and	The formal parameters **must include types** of the parameters. Example: `int x, real y`

	formal parameters ‘params’	Note that subprograms are evaluated **before** all other statements. A subprogram can be used anywhere in the global scope or inside any subprogram defined **after** it. Subprograms **cannot be nested**.
`conclude x`	Return value x from a function	Can only be inside a subprogram definition
`next`	Stop current iteration and proceed to the next one	Can only be inside an iteration structure
`stop`	Immediately exit from an iteration structure	Can only be inside an iteration statement
`nil`	Equivalent to NULL in C and None in Python.	Cannot be assigned to **any** variable; Can only be used in a conclude statement; It is of the type **void**
`yes` `no`	Reserved words for the Boolean values True and False	

Built-in Subprograms

To help programmers, Prodo has some built-in subprograms implemented in the *Prodo Python Library*. Note that these subprograms are not part of the functional specifications of the language.

`void write(str x)`

prints a string *x* onto the system’s standard output stream (no new line)

`str read()`

reads a line from the system’s standard input stream, removes the trailing new line, and returns the input as a string.

`void nl()`

prints a new line onto the system’s standard output stream

`int length(array x)`

returns the length of *x*

`array affix(array x, <?> e)`

returns a new array with the element *e* appended to the end. This function is defined for all types of *e* (int, real, str, bool, void, array, structure). The array is NOT modified.

`array f_read(str filename)`

opens the file with filename “filename”, reads its contents, and returns an array of the lines inside the file.

`void f_write(str filename, str contents)`

opens the file with filename “filename”, or creates it if it doesn’t exist. Then it writes “contents” into the file, overwriting any previous content.